Санкт-Петербургский государственный университет
Информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Реализация конечных автоматов на функциональных языках программирования

Ян Малаховски

2009

В работе рассматривается вопрос о реализации конечных автоматов на функциональных языках программирования. Автору не удалось найти каких-либо исследований на данную тему, однако он уверен, что использование функционального подхода при реализации конечных автоматов имеет свои преимущества, например, строгие проверки на наличие ошибок, компактный исходный код, возможность четкого разделения функциональной, императивной и автоматной частей программы.

Содержание

1 Введение

В данной работе описаны несколько методов реализации конечных автоматов в функциональных языках программирования. Работа носит практический характер и предлагает по новому взглянуть на привычные вещи.

В качестве языка программирования при реализации примеров в этой работе используется *Haskell*, так как он достаточно популярен в академической среде; в отличие от иных подобных языков, близок к обычному "типизированному лямбда исчислению" (отсутствие побочных эффектов); не нарушает функциональные концепции для осуществления ввода-вывода; имеет несколько качественных компиляторов.

В этой работе не проводится обучение автоматному или функциональному программированию. Она содержит лишь небольшое введение в данные области, которое необходимо для понимания излагаемого материала. Ознакомится с автоматным программированием можно в книге [1], а в качестве введения в функциональное программирование можно использовать [2] и, более старое и не столь актуальное, [3]. Прекрасным вводным курсом в язык *Haskell* может служить статья [4].

2 Введение в предмет

Этот раздел является введением в автоматное и функциональное программирование для тех, кто с ними мало или вообще не знаком.

Следующий подраздел представляет собой цитаты из [1, стр. 8], которые требуются для понимания данной работы. Во втором подразделе частично цитируется и используются материалы из лекций по функциональному программированию (автор не известен).

2.1 Автоматное программирование

Базовым понятием автоматного программирования является *состояние*. Понятие состояния в том смысле, как оно используется в данной парадигме, было введено А. Тьюрингом. Это понятие с успехом применяется во многих развитых областях науки, например, в теории управления и теории формальных языков.

Основное свойство состояния системы в момент времени t_0 заключается в отделении будущего ($t > t_0$) от прошлого ($t < t_0$), в том смысле, что текущее состояние несет в себе всю информацию о прошлом системы, необходимую для определения ее реакции на любое входное воздействие, формируемое в момент t_0. Состояние можно рассматривать как особую величину, которая в неявной форме объединяет все входные воздействия прошлого, влияющие на реакцию сущности в настоящий момент времени.

Понятие *входное воздействие* также является одним из базовых для автоматного программирования. Чаще всего, входное воздействие – это вектор. Его компоненты подразделяются на *события* и *входные переменные* в зависимости от смыс-

ла и механизма формирования. Совокупность конечного множества состояний и конечного множества входных воздействий образует *(конечный) автомат без выходов*. Такой автомат реагирует на входные воздействия, определенным образом изменяя текущее состояние. Правила, по которым происходит смена состояний, называют *функцией переходов* автомата.

То, что в автоматном программировании собственно и называется (конечным) автоматом (рис. 1), получается, если соединить понятие автомата без выходов с понятием *выходного воздействия*. Такой автомат реагирует на входное воздействие не только сменой состояния, но и формированием определенных значений на выходах. Правила формирования выходных воздействий называют *функцией выходов* автомата.

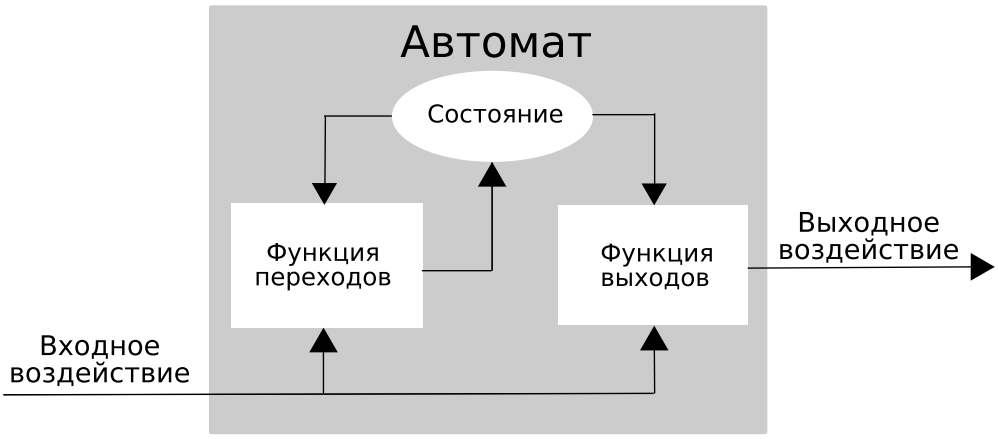

Рис. 1: Конечный автомат

2.2 Функциональное программирование

Программы на традиционных языках программирования, таких как, например *C* и *Pascal*, состоят из последовательности модификаций значений некоторого набора переменных. Если рассматривать только процесс вычисления выходных переменных по значениям входных переменных и не рассматривать операции ввода-вывода, то можно сказать, что до начала выполнения программы состояние (как совокупность входных, внутренних и выходных переменных программы) имеет некоторое начальное значение σ_0 (в котором представлены входные значения программы), а после завершения программы состояние имеет новое значение, включающее себя «результат» работы программы (выходные значения). Во время исполнения каждая команда изменяет состояние, а следовательно, оно проходит через некоторую конечную последовательность значений:

$$\sigma_0 \to \sigma_1 \to ... \to \sigma_n = \sigma'$$

Модификация состояний производится про помощи команд *присваивания* ($v = e$), следующих одна за другой, а также циклов (for, while, ...) и операций ветвления (if, switch, ...). Такой стиль программирования называют *императивным* или *процедурным*.

Функциональное программирование представляет парадигму, в корне отличную от представленной выше. Функциональная программа представляет собой некоторое выражение (в математическом смысле), а выполнение программы означает вычисление значения этого выражения (в данном случае слово «вычисление» вовсе не означает, что операции производятся только над числами). С учетом приведенных выше обозначений, считая, что результат работы программы зависит только от значений ее входных переменных, можно сказать, что результирующее состояние (или любое промежуточное) представляет собой функцию от начального состояния — $\sigma' = f(\sigma)$.

В функциональном программировании используется именно эта точка зрения: программа представляет собой выражение, соответствующее функции f, а сами функциональные языки программирования поддерживают построение таких выражений, предоставляя широкий выбор соответствующих языковых конструкций.

При сравнении функционального и императивного подхода к программированию можно отметить следующие свойства функциональных программ:

- Функциональные программы не используют переменные в том смысле, в котором это слово употребляется в императивном программировании. В частности, в функциональных программах не используется оператор присваивания.
- Как следствие из предыдущего пункта, в функциональных программах нет циклов.
- Выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.
- Функциональные программы используют функции гораздо более замысловатыми способами, чем при использовании императивных (традициооных) языков программирования. Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и даже в общем случае проводить вычисления, результатом которого будет функция.
- Вместо циклов функциональные программы широко используют рекурсивные функции.

3 Первый взгляд на проблему

3.1 Проблемы реализации

На первый вгляд, написание функциональных программ в автоматном стиле («программирование от состояний») должно представлять некоторые трудности:

- При рассмотрении некоторого лямбда-выражения (функции), изменяющего состояние системы $\sigma_n \to \sigma_m$, нельзя говорить о промежуточных состояниях ($n < k < m$, где k — номер промежуточного состояния), поскольку, в общем случае, невозможно указать место в коде программы, где это состояние принималось бы системой.

- Существует опасность резкого увеличения размеров исходного кода при росте размера вектора состояния системы. Следует упомянуть, что с этой проблемой сталкиваются и программисты на традиционных языках, однако в функциональном случае все усугубляется еще и тем, что каждой функции, как правило, приходится работать со всем вектором состояния, поскольку в функциональном программировании не существует прямого аналога традиционному представлению глобальных переменных или объектов объектно-ориентированного программирования.

Однако, как будет показано далее, обе проблемы вполне преодолимы, первая — при помощи определенного способа организации вычислений, а вторая — использованием «синтаксического сахара» (конструкции, эквивалентные уже существующим в языке, но упрощающие написание и чтение исходного кода), предлагаемого языком *Haskell*. Кроме того, при соблюдении некоторых принципов программирования появляется возможность валидации реализации автомата самим компилятором. Это является весьма труднодостижимым при приминении традиционных языков программирования.

3.2 Пример 1. Счетный триггер

Пусть требуется реализовать «счетный триггер», реагирующий на события. Другими словами, требуется построить конечный автомат с двумя состояниями, кодируемыми нулем и единицей, который управляется кнопкой. Каждое нажатие кнопки порождает событие x. К выходу z конечного автомата подключена лампа. Каждое событие x переводит автомат в состояние *(1 - y)*, где y — текущее состояние. В состоянии *0* выход z равен нулю, а в состоянии *1* — единице. Соответственно, каждое нажатие кнопки будет приводить то к включению лампы, то к ее выключению.

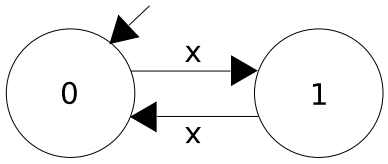

Рис. 2: Диаграмма состояний счетного триггера, 0 — состояние с выключенной лампой, 1 — с включенной, x — нажатие кнопки

3.3 Несколько возможных реализаций

Рассмотрим несколько примеров реализации описанного примера (листинги 1 — 3). Начальное состояние конечного автомата — 0 (лампа выключена), потом кто-то нажимает на кнопку (лампа должна включиться), программа печатает состояние конечного автомата и завершается.

Листинг 1: Реализация счетного триггера на *C*

```
#include <stdio.h>

// Событие: <<Нажатие на кнопку>>.
#define EVENT_BUTTONCLICK 0

// Состояния: <<Выключено>> и <<Включено>>.
#define LAMPOFF 0
#define LAMPON 1

// Переменная состояния, начальное состояние — LAMPOFF.
int current_state = LAMPOFF;

// Функция переходов.
void got_event(int event)
{
    if (event == EVENT_BUTTONCLICK)
        switch (current_state)
        {
            case LAMPOFF: current_state = LAMPON; break;
            case LAMPON: current_state = LAMPOFF; break;
        }
}

// Функция, вызываемая системой.
int main()
{
    got_event(EVENT_BUTTONCLICK);
    printf("%i", current_state);
}
```

Листинг 2: Реализация счетного триггера на *C* с использованием *enum*

```
#include <stdio.h>

// Событие: <<Нажатие на кнопку>>.
enum event {EVENT_BUTTONCLICK};

// Состояния: <<Выключено>> и <<Включено>>.
enum state {LAMPOFF, LAMPON};

// Переменная состояния, начальное состояние — LAMPOFF.
int current_state = LAMPOFF;
```

```
// Функция переходов.
void got_event(int event)
{
    if (event == EVENT_BUTTONCLICK)
        switch (current_state)
        {
            case LAMPOFF: current_state = LAMPON; break;
            case LAMPON: current_state = LAMPOFF; break;
        }
}

// Функция, вызываемая системой.
int main()
{
    got_event(EVENT_BUTTONCLICK);
    printf("%i", current_state);
}
```

Листинг 3: Реализация счетного триггера на *Haskell*

```
-- Событие: <<Нажатие на кнопку>>.
data Event = ButtonClick deriving Show

-- Состояния: <<Выключено>> и <<Включено>>.
data State = LampOff | LampOn deriving Show

-- Функция переходов.
gotEvent :: State -> Event -> State
gotEvent state event = case event of
            ButtonClick -> case state of
                LampOff -> LampOn
                LampOn  -> LampOff

-- Начальное состояние -- LampOff.
nstate = LampOff

-- Функция, вызываемая системой.
main = putStrLn (show (gotEvent nstate ButtonClick))
```

3.4 Сравнение

Рассмотрим листинг 1. Если опустить строку с *#include*, то код начинается с объявления констант. Чем больше событий (разновидность входных воздействий), внутренних и выходных переменных у автомата — тем больше констант. В листинге 2 этот недостаток устранен при помощи конструкции *enum*. Однако и в этом случае, и события, и состояния автомата являются числами. Более того, компилятор по умолчанию нумерует элементы *enum* с нуля, так что вовсе не сложно где-нибудь в программе сравнить событие с состоянием, что является ошибкой.

Реализация на языке *Haskell* использует алгебраические типы данных и лишена этих недостатков. Состояния и события — это разные типы, и при попытке их

сравнения компилятор укажет на неверно типизованное выражение. Подобный результат можно получить и в императивных языках, реализовав алгебраические типы данных, которые по умолчанию присутствуют в языке *Haskell*. Однако при большом числе автоматов число описанных вручную типов данных может исчисляться десятками, что ведет к "раздуванию" кода, или к разработке отдельного приложения для автоматической генерации таких конструкций.

Таким образом, даже на таком простом примере можно заметить преимущества реализации конечных автоматов на *Haskell*, получаемые за счет развитой системы типов этого языка. Поэтому подобный подход к описанию событий и состояний будет применяться и далее.

4 Рекурсивная реализация конечных автоматов

4.1 Рекурсия

Для реализации конечного автомата необходим некий механизм смены состояний. В императивном программировании для этих целей используются циклы, а в типизированном функциональном программировании циклов нет. Сохранение состояния системы в императивных языках программирования, как правило, осуществляется за счет использования побочных эффектов (глобальные переменные, поля классов), а в "чистой лямбде" (и в языке *Haskell* тоже) функции не могут иметь побочных эффектов. Таким образом, автоматы на функциональных языках должны строиться на других принципах.

Поэтому в настоящей работе предлагается представлять конечный автомат, как кортеж (вектор) всех его переменных, а в качестве механизма смены состояний использовать рекурсию. Тогда функция переходов должна принимать в качестве параметров кортеж старого состояния и входные воздействия, а возвращать кортеж нового состояния и как-либо генерировать выходные воздействия. Пример подобной реализации функции переходов приведен в листинге 3. В нем вместо кортежа переменных используется одна переменная состояния, а в качестве входного воздействия — только единственное событие.

Для простоты будем считать, что на вход автомата поступают только события. При этом представим последовательность этих событий в виде списка и будем последовательно применять функцию переходов к каждому элементу списка и кортежу текущего состояния.

4.2 Развитие реализации счетного триггера

Пользуясь вышеописанными положениями, разовьем пример реализации счетного триггера на *Haskell*, добавив возможность выполнять переход по списку событий.

Листинг 4: Развитие реализации счетного триггера

```
-- Событие: <<Нажатие на кнопку>>.
data Event = ButtonClick deriving Show
```

```
-- Состояния: <<Выключено>> и <<Включено>>.
data State = LampOff | LampOn deriving Show

-- Функция переходов.
gotEvent :: State -> Event -> State
gotEvent state event = case event of
            ButtonClick -> case state of
                LampOff -> LampOn
                LampOn  -> LampOff

-- Функция, применяющая список событий к
-- начальному состоянию, для получаения
-- результирующего состояния.
applyEvents :: State -> [Event] -> State
applyEvents state []     = state
applyEvents state (e:es) =
    applyEvents (gotEvent state e) es

-- Начальное состояние -- LampOff.
nstate = LampOff

-- Функция, вызываемая системой.
main = putStrLn $ show $
    applyEvents nstate [ButtonClick, ButtonClick]
```

В приведенном листинге введена новая функция *applyEvents*, принимающая в качестве параметров: состояние, к которому необходимо применить список событий, и сам список событий. Если этот список пуст, то функция возвращает переданное ей состояние, иначе она вызывает себя рекурсивно, но в качестве нового списка событий использует хвост текущего, а в качестве нового состояния — результат применения функции переходов к голове списка и текущему состоянию.

Отметим, что при ручном программировании функций переходов конечных автоматов программистам на императивных языках часто приходится следить за структурой выражений, используемых для вычисления новых переменных состояния, поскольку смена их значений должна происходить не последовательно, а одновременно во всей системе. Поэтому в подобных реализациях применяются промежуточные переменные: сначала программа считает значения изменяющихся переменных и записывает их в промежуточные переменные, а затем следует блок присваиваний значений промежуточных переменных переменным состояния, за исключением, быть может, последней переменной. В то же время, программист на функциональном языке избавлен от необходимости следить за подобными вещами вручную. Одновременность перехода обеспечивается за счет использования кортежей или структур.

4.3 Исключение дублирования кода

Если бы листинг 4 содержал более одного автомата, то пришлось бы писать несколько аналогичных *applyEvents* функций. Поэтому можно выделить общую

часть кода этих функций в своего рода библиотечный код (листинг 5), пригодный при написании конечных автоматов.

Листинг 5: Выделение общих частей

```
-------------------------------------------------------------
-- Своего рода библиотечный код.
-------------------------------------------------------------

-- Абстрактный тип функции переходов.
type SwitchFunc state event = state -> event -> state

-------------------------------------------------------------
-- applyEvents - Функция, применяющая список событий
-- к начальному состоянию автомата при помощи
-- функции переходов.
--
-- Аргументы:
-- 1) функция переходов
-- 2) начальное состояние
-- 3) список событий.
--
-- Результат: результирующее состояние автомата после
-- всех переходов, сделанных в соответствии со
-- списком событий.
-------------------------------------------------------------
applyEvents :: SwitchFunc state event
    -> state -> [event] -> state
-- Если список событий пуст, то возвращаем
-- последнее состояние,
applyEvents _ state [] = state
-- иначе обрабатываем одно событие и вызываемся
-- рекурсивно с хвостом списка событий.
applyEvents switchFunc state (event:eventsTail) =
    applyEvents switchFunc
        (switchFunc state event) eventsTail

-------------------------------------------------------------
-- Собственно программа.
-------------------------------------------------------------

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case event of
                    ButtonClick -> case state of
                        LampOff -> LampOn
                        LampOn  -> LampOff

-- Функция, вызываемая системой.
main = putStrLn $ show $ applyEvents
```

При этом отметим, что новая версия описанной функции *applyEvents*, является сверткой (одна из стандартных операций функционального программирования) списка событий по функции переходов.

В примерах исходного кода, приводимых ниже, описанная библиотечная часть дублироваться не будет.

4.4 Пример 2. Счетный триггер с четырьмя состояниями

Рассмотрим счетный триггер с четырьмя состояниями. Его отличие от предыдущего триггера заключается в том, что лампа будет включаться или выключаться только после отпускания кнопки, а повторное нажатие или отпускание кнопки не будет производить ни какого эффекта.

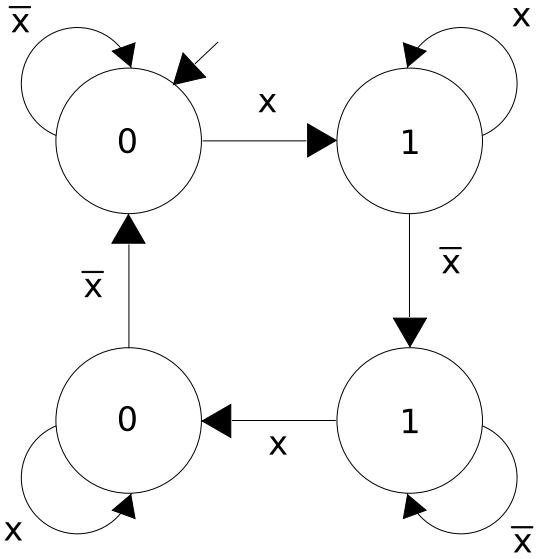

Рис. 3: Диаграмма переходов счетного триггера с четырьмя состояниями, 0 — состояния с выключенной лампой, 1 — с включенной, x — нажатие кнопки

Листинг 6: Счетный триггер с четырьмя состояниями

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case event of
    ButtonDown -> case state of
        LampOffButtonUp -> LampOffButtonDown
        LampOnButtonUp -> LampOnButtonDown
        _ -> state
    ButtonUp -> case state of
        LampOffButtonDown -> LampOnButtonUp
        LampOnButtonDown -> LampOffButtonUp
        _ -> state

-- Функция, вызываемая системой.
main = putStrLn $ show $ applyEvents triggerSwitchFunc
    LampOffButtonUp
    [ButtonDown, ButtonUp, ButtonDown, ButtonDown, ButtonUp]
```

Заметим, что в этом примере у функции переходов, в которой сначала проверяется событие, а потом уже состояние, код получается короче (в чем не трудно самостоятельно убедиться), чем у функции с обратным порядком проверок.

5 Реализация вложенности

5.1 Вложенность

До этого речь шла только о представлении конечных автоматов, которые реагируют на внешние воздействия только сменой собственных состояний, а выходным воздействием была печать на экран результирующего состояния. Однако кортеж состояния системы может содержать описание состояний не только одного автомата. Таким образом можно реализовывать вложенные конечные автоматы.

5.2 Пример 3. Устройство с меню

В качестве задачи для реализации системы со вложенностью рассмотрим чрезвычайно упрощенную модель цифрового устройства с четырьмя кнопками на корпусе и экраном. Устройство может быть включено, отображать меню, состоящее из некоторого списка элементов, и быть выключено, причем выключенное устройство запоминает состояние меню и при следующем входе в него отображает сохраненную позицию курсора.

Следовательно, имеется два конечных автомата (устройство и меню), причем второй вложен в первый. Автомат, описывающий устройство, имеет три состояния: «Выключено», «Включено» и «Меню», а автомат, описывающий меню, — два: «Выбран первый пункт меню» и «Выбран второй пункт меню». Панель устройства содержит четыре кнопки: «Включить-выключить», «Меню», «Вверх», «Вниз». Графы переходов системы представлены на рис. 4.

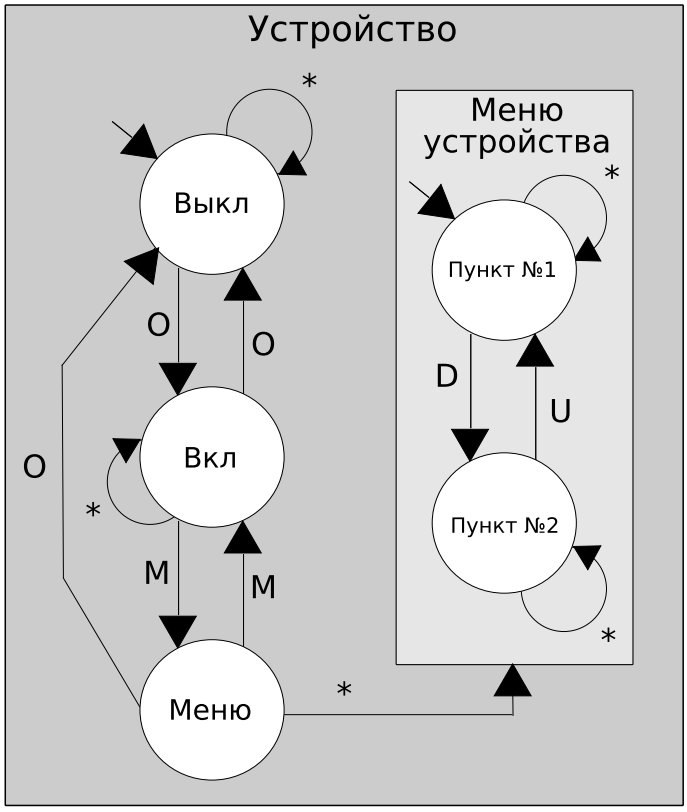

Рис. 4: Диаграммы состояний устройства и его меню. O, M, U, D — кнопки включения-выключения, меню, вверх и вниз соответственно

5.3 Два варианта реализации

Два нижеследующих листинга демонстрируют реализацию данной системы.

Листинг 7: Модель устройства, реализованная при помощи кортежей

```
-- Типы событий и состояний для устройства.
data DeviceEvent = OnOffButton
                 | MenuButton
```

```
                  | UpButton
                  | DownButton deriving Show
data DeviceState = DeviceIsOff
                  | DeviceIsOn
                  | DeviceIsInMenu deriving Show

-- Типы событий и состояний для меню.
data MenuEvent = DeviceButton DeviceEvent deriving Show
data MenuState = MenuIsInPositionOne
                  | MenuIsInPositionTwo deriving Show

-- Функция переходов для устройства.
deviceSwitchFunc state@(dstate, mstate) event = case dstate of
    DeviceIsOff -> case event of
        OnOffButton -> (DeviceIsOn, mstate)
        _ -> state
    DeviceIsOn -> case event of
        OnOffButton -> (DeviceIsOff, mstate)
        MenuButton -> (DeviceIsInMenu, mstate)
        _ -> state
    DeviceIsInMenu -> case event of
        OnOffButton -> (DeviceIsOff, mstate)
        MenuButton -> (DeviceIsOn, mstate)
        _ -> (dstate, menuSwitchFunc mstate (DeviceButton event))

-- Функция переходов для меню.
menuSwitchFunc mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = putStrLn $ show $ applyEvents
    deviceSwitchFunc (DeviceIsOff, MenuIsInPositionOne)
        [OnOffButton, MenuButton, DownButton, OnOffButton]
```

Листинг 8: Модель устройства, реализованная при помощи структур

```
-- Типы событий и состояний для устройства.
data DeviceEvent = OnOffButton
                  | MenuButton
                  | UpButton
                  | DownButton deriving Show
data DeviceState = DeviceIsOff
                  | DeviceIsOn
                  | DeviceIsInMenu deriving Show

-- Типы событий и состояний для меню.
data MenuEvent = DeviceButton DeviceEvent deriving Show
```

```
data MenuState = MenuIsInPositionOne
               | MenuIsInPositionTwo deriving Show

data SystemState = SystemState
    { dstate :: DeviceState
    , mstate :: MenuState } deriving Show

-- Функция переходов для устройства.
deviceSwitchFunc state event = case dstate state of
    DeviceIsOff -> case event of
        OnOffButton -> state { dstate = DeviceIsOn }
        _ -> state
    DeviceIsOn -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsInMenu }
        _ -> state
    DeviceIsInMenu -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsOn }
        _ -> state { mstate =
            (menuSwitchFunc (mstate state)
                (DeviceButton event)) }

-- Функция переходов для меню.
menuSwitchFunc mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = putStrLn $ show $ applyEvents
    deviceSwitchFunc
    ( SystemState
        { dstate = DeviceIsOff
        , mstate = MenuIsInPositionOne } )
    [OnOffButton, MenuButton, DownButton, OnOffButton]
```

Разница между первой и второй реализациями, как следует из названий листингов, заключается в том, что в первой система описана через кортеж, а во второй — через структуру. Для данной задачи разница размеров исходных кодов двух реализаций несущественна, однако для больших векторов состояний системы использование структур выглядит привлекательнее, поскольку для них компилятор автоматически генерирует функции извлечения элементов из вектора состояний.

6 Реализация с использованием монад

6.1 Монады

Одной из наиболее значимых встроенных конструкций языка *Haskell* являются *монады*. С их помощью, например, осуществляется ввод-вывод, который не нарушает функциональные основы языка. О монадах можно думать, как о способе объединения последовательности действий — в некотором смысле способ "эмулировать" императивное программирование в рамках функционального. Более подробно о монадах можно прочитать в работе [4] и в работе [5]. Из сказанного не следует, что монады невозможно реализовать в других функциональных языках программирования, однако в *Haskell* они используются повсеместно.

Поэтому логичным развитием идеи реализации конечных автоматов на *Haskell* является попытка построить монаду, удобную в использовании.

6.2 Пример 4. Реализация счетного триггера с использованием монады FSM

Снова обратимся к счетному триггеру на два состояния (рис. 2). Однако теперь реализуем этот конечный автомат с помощью монады, которую назовем *FSM*, реализовав небольшую "библиотеку".

Листинг 9: Реализация счетного триггера при помощи монады *FSM*

```
-----------------------------------------------------------
-- Своего рода библиотечный код.
-----------------------------------------------------------

-- Абстрактный тип функции перехода.
type SwitchFunc state event = state -> event -> state

-- Тип монадического преобразования состояния.
newtype FSM state a = FSM ( state -> (state, a) )

-- Реализация функций класса Monad.
instance Monad (FSM state) where
    -- (>>=) :: FSM state a ->
    -- (a -> FSM state b) -> FSM state b
    (FSM first) >>= second =
        FSM ( \s0 -> let (s1, a) = first s0
                         (FSM q) = second a in q s1 )

    -- return :: a -> FSM state a
    return a = FSM ( \s -> (s, a) )

-- Взятие текущего состояния.
getState :: FSM state state
getState = FSM ( \was -> (was, was) )

-- Установка текущего состояния.
```

```
setState :: state -> FSM state ()
setState state = FSM ( \s -> (state, ()) )

-- Функция для применения монады.
-- Передает начальное состояние первым аргументом.
applyFSM :: s -> FSM s a -> (s, a)
applyFSM s (FSM p) = p s

-- Переход по событию.
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    setState (switchFunc st event)

-- Переход по списку событий.
applyEvents :: SwitchFunc state event -> [event] -> FSM state ()
applyEvents _ [] = return ()
applyEvents switchFunc (event:eventsTail) = do
    applyEvent switchFunc event
    applyEvents switchFunc eventsTail

-----------------------------------------------------------
-- Собственно программа.
-----------------------------------------------------------

data Event = ButtonClick deriving Show
data State = LampOff | LampOn deriving Show

gotEvent :: State -> Event -> State
gotEvent state event = case event of
            ButtonClick -> case state of
                LampOff -> LampOn
                LampOn  -> LampOff

nstate = LampOff
main = putStrLn $ show $ fst $
    applyFSM nstate $
        applyEvents gotEvent [ButtonClick, ButtonClick]
```

Приведенная реализация библиотечного кода для монады *FSM* содержит следующие функции для выполнения операций над состояниями:

- *getState* – получает текущее значение состояния конечного автомата;
- *setState* – устанавливает текущее состояние конечного автомата в определенное значение;
- *applyEvent* – производит переход по событию при помощи некоторой функции переходов;
- *applyEvents* – производит переход по списку событий;
- *applyFSM* – вычисляет результат выполнения монады *FSM*.

По сравнению с листингом 5 объем "библиотечного" исходного кода значительно увеличился. С другой стороны, появилась возможность запоминать состояние автомата в некоторый момент времени при помощи функции *getState* и устанавливать его в другом месте при помощи функции *setState*. Таким образом, можно, например, заставить конечный автомат перейти "назад во времени".

6.3 Пример 5. Реализация счетного триггера с использованием монады State

Стандартная библиотека языка *Haskell* содержит монаду *State*, которая предоставляет обобщенный механизм манипуляции состояниями. В частности, эта монада может использоваться для реализации конечных автоматов.

Монада *State* предоставляет следующие функции для манипуляции состояниями:

- *get* – получает текущее значение состояния;
- *put* – устанавливает текущее состояние;
- *runState* – вычисляет результат выполнения монады *State*.

В приведенном ниже листинге 10 функции *getState*, *setState*, *applyEvent*, *applyEvents*, *applyFSM* (интерфейс монады *FSM*) реализованы при помощи функций, предоставляемых стандартной монадой *State*.

Листинг 10: Реализация счетного триггера при помощи монады State

```
-----------------------------------------------------------
-- Своего рода библиотечный код.
-----------------------------------------------------------
import Control.Monad.State as S

-- Абстрактный тип функции перехода.
type SwitchFunc state event = state -> event -> state

-- Тип монадического преобразования сотояния.
type FSM state a = S.State state a

-- Взятие текущего состояния.
getState :: FSM state state
getState = get

-- Установка текущего состояния.
setState :: state -> FSM state ()
setState state = put state

-- Функция для применения монады.
-- Передает начальное состояние первым аргументом.
applyFSM :: s -> FSM s a -> (s, a)
applyFSM s p = (\(a,b) -> (b,a)) (runState p s)
```

```
-- Переход по событию.
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    setState (switchFunc st event)

-- Переход по списку событий.
applyEvents :: SwitchFunc state event -> [event] -> FSM state ()
applyEvents _ [] = return ()
applyEvents switchFunc (event:eventsTail) = do
    applyEvent switchFunc event
    applyEvents switchFunc eventsTail

-----------------------------------------------------------
-- Собственно программа.
-----------------------------------------------------------

data Event = ButtonClick deriving Show
data AState = LampOff | LampOn deriving Show

gotEvent :: AState -> Event -> AState
gotEvent state event = case event of
            ButtonClick -> case state of
                LampOff -> LampOn
                LampOn  -> LampOff

nstate = LampOff
main = putStrLn $ show $ fst $
    applyFSM nstate $
        applyEvents gotEvent [ButtonClick, ButtonClick]
```

Таким образом, предложенная монада *FSM* из листинга 9 представляет собой альтернативную реализацию *State*, дополненную операциями специфичными именно при реализации конечных автоматов.

6.4 Проблемы и применимость подхода

Проблемы реализации с использованием монад состоят в том, что в них плохо вписываются выходные воздействия.

Поскольку в общем случае, и входные, и выходные воздействия могут иметь тип *IO* (монада, осуществляющая ввод-вывод), то и монада *FSM*, должна иметь возможность осуществлять ввод-вывод, а следовательно, в ней должна иметься возможность "вкрапления" конструкций, имеющих тип *IO*. Поскольку сама монада *FSM* занимается только преобразованием состояния конечного автомата, то вполне логичным "вкраплением" *IO* является полный отказ от самой монады *FSM*, так как этом случае она не предоставляет ничего нового, кроме «синтаксического сахара». Иначе говоря, в этом случае, она не добавляет возможностей валидации, а лишь предоставляет потенциальную возможность записывать длинные цепочки переходов конечного автомата более коротким исходным кодом, чем это требовалось в подходах без использования монад.

С другой стороны, если входные воздействия представимы без *IO*, а выходные воздействия осуществляются только внутри монады *FSM* и/или являются ее результатом, то применение монады *FSM* вполне оправдано. Например, при реализации парсеров на *Haskell* используются конструкции, схожие с *FSM* [6].

7 Рекурсивная реализация с выделением функции выходов

7.1 Суть подхода

Оценив применимость подхода с монадами, попробуем отделить выходные воздействия от функции переходов. Таким образом, автомат в целом будет записан в классическом функциональном стиле, и лишь функция выходов будет иметь доступ к *IO* монаде.

Иначе говоря, модель, представленная в начале работы (см. рис. 1), будет переведена в полный ее функциональный аналог.

7.2 Пример 6. Реализация счетного триггера с выделением функции выходов

В качестве задачи для реализации снова рассмотрим счетный триггер с двумя состояниями (см. рис. 2).

Листинг 11: Выделение функции выходов

```
-----------------------------------------------------------
-- Своего рода библиотечный код.
-----------------------------------------------------------

-- Абстрактный тип функции переходов.
type SwitchFunc state event = state -> event -> state
-- Абстрактный тип функции выходов.
type OutFunc state event = state -> event -> IO ()

-----------------------------------------------------------
-- applyEvents - Функция, применяющая список событий
-- к начальному состоянию автомата при помощи
-- функции переходов и генерирующая выходные
-- воздействия при помощи функции выходов.
--
-- Аргументы:
-- 1) функция переходов
-- 2) функция выходов
-- 3) начальное состояние
-- 4) список событий.
--
-- Результат: результирующее состояние автомата
-- и выходные воздействия, полученные после
-- всех переходов, сделанных в соответствии
```

```
-- со списком событий.
-----------------------------------------------------------
applyEvents :: SwitchFunc state event ->
    OutFunc state event -> state -> [event] -> IO state
-- Если список событий пуст, то возвращаем
-- последнее состояние,
applyEvents _ _ state [] = return state
-- иначе обрабатываем одно событие и вызываемся
-- рекурсивно с хвостом списка событий.
applyEvents switchFunc outFunc state (event:eventsTail) = do
    outFunc state event
    applyEvents switchFunc outFunc newstate eventsTail
    where newstate = switchFunc state event

-----------------------------------------------------------
-- Собственно программа.
-----------------------------------------------------------

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case event of
                    ButtonClick -> case state of
                        LampOff -> LampOn
                        LampOn  -> LampOff

-- Функция выходов.
triggerOutFunc state event = putStrLn $ show $ state

-- Функция, вызываемая системой.
main = applyEvents triggerSwitchFunc
            triggerOutFunc LampOff [ButtonClick, ButtonClick]
```

Как следует из листинга, в данной модели представления конечных автоматов только функции с типом *OutFunc* содержат в себе *IO*, а функция переходов с типом *SwitchFunc* записана в классическом функциональном стиле.

7.3 Функция выходов, как часть функции переходов

Однако в рассмотренной в предыдущем примере интерпретации модели конечного автомата существуют и недостатки.

- В задачах чуть более сложных, нежели приведенная, функциям выходов придется частично повторять работу функции переходов.
- Разделение одного конечного автомата на две части несет потенциальные проблемы читаемости кода, поскольку приходится “метаться” между функцией переходов и функцией выходов для восстановления диграммы состояний по уже написанной программе.

Таким образом, логичным решением является перенос части (или всей целиком) функции выходов в функцию переходов, у которой есть доступ, как в прошлому состоянию системы, так и к формируемому.

7.4 Пример 7. Перенос функции выходов в функцию переходов

Листинг 12: Перенос функции выходов в функцию переходов

```
-----------------------------------------------------------
-- Своего рода библиотечный код.
-----------------------------------------------------------

-- Абстрактный тип функции переходов, возвращающей
-- новое состояние и список выходных воздействий.
type SwitchFunc state input output = state -> input
    -> (state, [output])

-----------------------------------------------------------
-- applyEvents - Функция, применяющая список событий
-- к начальному состоянию автомата при помощи
-- функции переходов.
--
-- Аргументы:
-- 1) функция переходов
-- 2) начальное состояние
-- 3) список событий.
--
-- Результат: результирующее состояние автомата
-- и выходные воздействия, полученные после
-- всех переходов, сделанных в соответствии
-- со списком событий.
-----------------------------------------------------------
applyEvents :: SwitchFunc state input output
    -> state -> [input] -> (state, [output])
-- В самом начале список выходных воздействий пуст
applyEvents switchFunc state events =
    applyEvents_ switchFunc state [] events

-- Если список событий пуст, то возвращаем
-- результирующее состояние и накопленные
-- выходные воздействия,
applyEvents_ _ state output [] = (state, output)
-- иначе обрабатываем одно событие и вызываемся
-- рекурсивно с хвостом списка событий,
-- накопив еще одно выходное воздействие.
applyEvents_ switchFunc state output (event:eventsTail)=
    applyEvents_ switchFunc newstate
        (output ++ newoutput) eventsTail
    where (newstate, newoutput) = switchFunc state event
```

```
-- Типы событий, состояний и выходных воздействий
-- для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show
data TriggerOut = OffTheLamp
                | OnTheLamp deriving Show

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case event of
    ButtonDown -> case state of
        LampOffButtonUp ->
            (LampOffButtonDown, [putStrLn "Nothing"])
        LampOnButtonUp ->
            (LampOnButtonDown, [putStrLn "Nothing"])
        _ -> (state, [])
    ButtonUp -> case state of
        LampOffButtonDown ->
            (LampOnButtonUp, [putStrLn $ show $ OnTheLamp])
        LampOnButtonDown ->
            (LampOffButtonUp, [putStrLn $ show $ OffTheLamp])
        _ -> (state, [])

-- Функция, вызываемая системой.
main = do
    putStrLn $ show $ s
    sequence_ o
    where (s, o) =
        applyEvents triggerSwitchFunc LampOffButtonUp
            [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
```

Преимуществом данного подхода является его гибкость. Например, список выходных воздействий может состоять из элементов специально введенного алгебраического типа данных, а перевод элементов этого типа в собственно выходные воздействия может производиться уже вне функции переходов. Таким образом, в данном подходе можно совместить преимущества предыдущего подхода с удобством программирования. При этом не теряется возможность отдать на валидацию компилятору как можно больше кода.

8 Заключение

В представленной работе были предложены несколько наиболее удачных, по мнению автора, подходов к реализации конечных автоматов на языке *Haskell*. Однако остаются не рассмотренными вопросы возможности более строгой типизации реализуемых конечных автоматов, а использование зависимой системы типов вообще не изучено.

Список литературы

[1] Н. И. Поликарпова, А. А. Шалыто. *Автоматное программирование. Учебно-методическое пособие.* СПБГУ ИТМО, Санкт-Петербург, 2007. http://is.ifmo.ru/books/_umk.pdf

[2] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, New York, 1998.

[3] A. Davie. *Introduction to Functional Programming System Using Haskell.* Cambrige University Press, 1992.

[4] Paul Hudak, John Peterson and Joseph H. Fasel. *A Gentle Introduction to Haskell 98.* http://www.haskell.org/tutorial/

[5] Евгений Кирпичев. *Монады.* RSDN Magazine, 2008, #3. http://www.rsdn.ru/

[6] Parsec monadic parser combinator library for Haskell http://www.haskell.org/haskellwiki/Parsec